Return To Agenda Page

Software and the Concurrency Revolution

Herb Sutter

Software Architect
Microsoft Developer Division

The Last Slide First

What you need to know about concurrency

It's here

parallelism has long been the "next big thing"

the future is now, and inescapable

It will directly affect the way we write software

the free lunch is over – for sequential apps

only apps with lots of latent concurrency get the free lunch back

(but responsiveness is the other reason to want concurrency)

languages won't be able to ignore it and stay relevant

The software industry has a lot of work to do,

and we suspect the HW industry vastly underestimates that

a generational advance >OO to move beyond "threads+locks"

key: incrementally adoptable extensions for existing languages

Concurrency

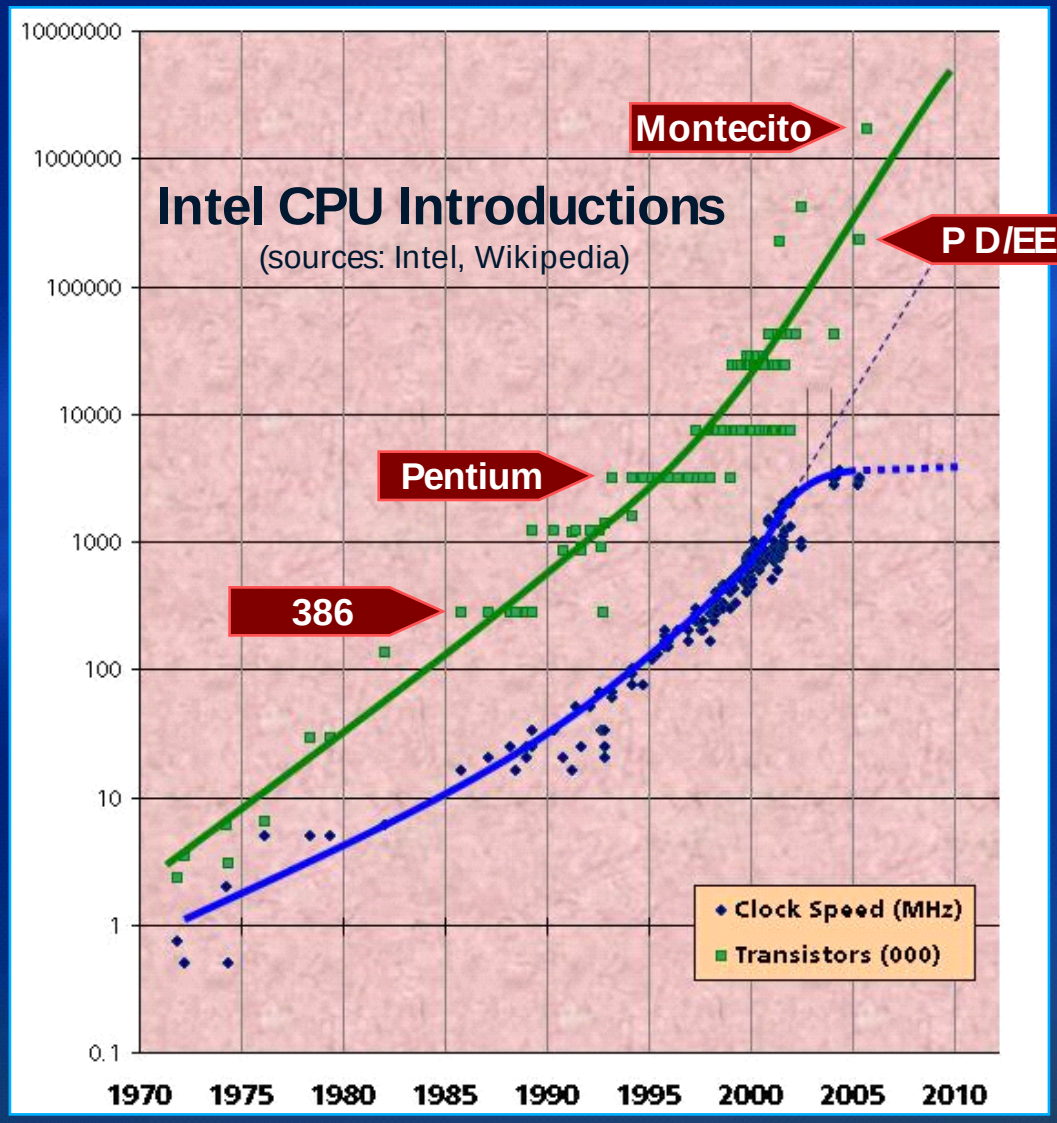
Truths

Consequences

Futures

The Free Lunch Is Over

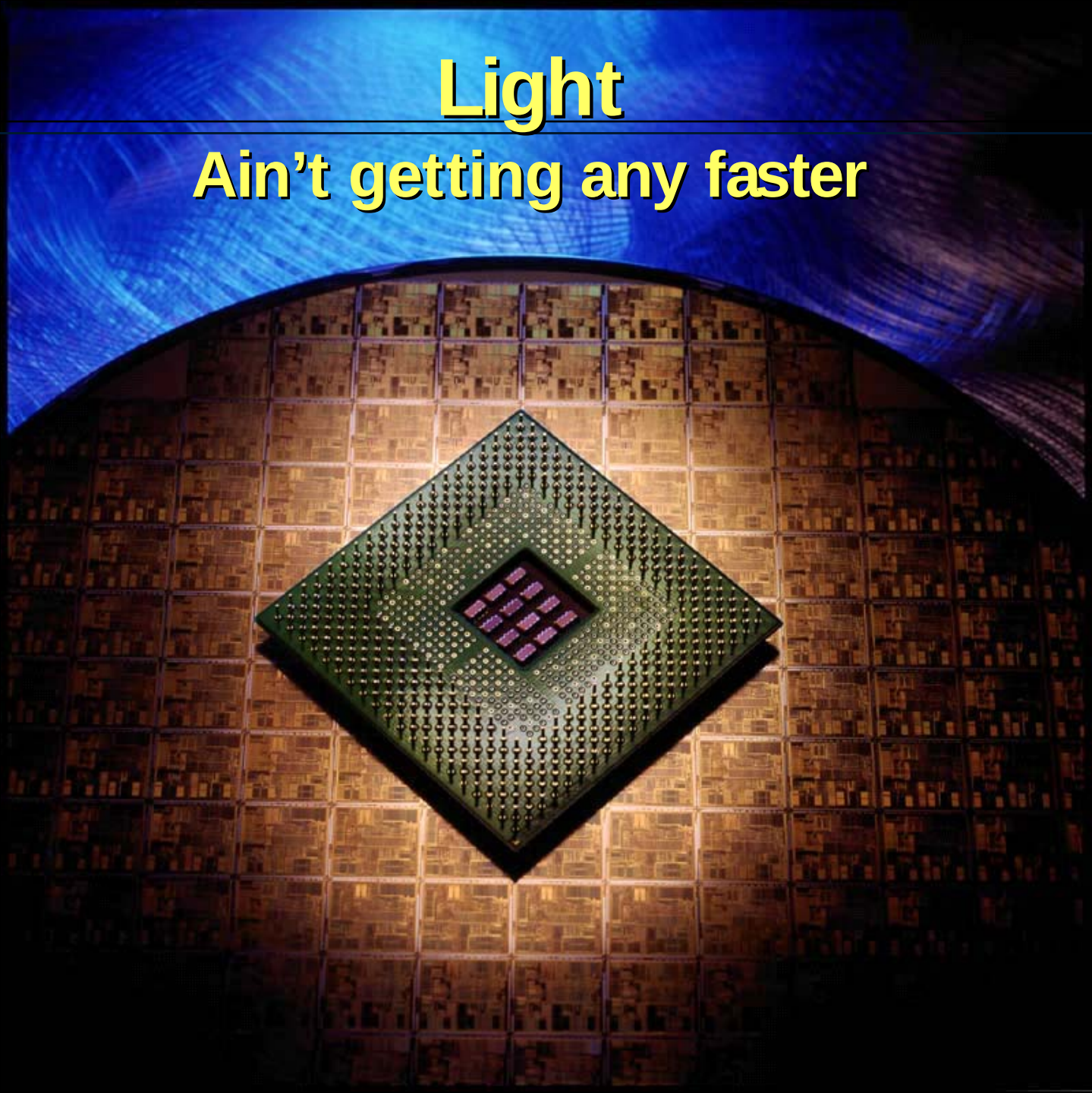
Unless we reenable it…



- Nonconcurrent apps aren't going to get faster (expect some regressions). We need killer apps with lots of latent parallelism.
- The state of concurrency in the software industry is terrible. Nobody does concurrency well for mainstream languages and platforms. The world's best frameworks are just getting away with it.
- A generational advance >OO is necessary to get above the "threads+locks" programming model.

Light

Ain’t getting any faster



A Software Revolution

Motivating an "OO for concurrency"

Concurrency is likely to be more disruptive than OO

Languages can't ignore it

languages could ignore OO and remain relevant (e.g., C)

today's languages will be forced to add direct support for concurrency, or be marginalized to non-demanding apps

It's demonstrably harder

e.g., analysis that is routine for sequential programs is provably undecidable for concurrent programs

We need higher-level abstractions for mainstream languages

"threads + locks" ≡ structured programming

necessary new abstractions ≡ objects

The Issue Is (Mostly) On the Client

What's "already solved" and what's not

"Solved": Server apps (e.g., database servers, web services)

typical to execute many copies of the same code

shared data usually via structured databases

with some care, "concurrency problem is already solved" here

Not solved: Typical client apps

highly atypical to execute many copies of the same code

shared data in memory, unstructured and promiscuous

legacy requirements to run on a given thread (e.g., GUI)

Surveying the New World

	Sequential Programs	Concurrent Programs
Behavior	Deterministic	Nondeterministic
Memory	Stable	In flux (unless private, read-only, or protected by lock)
Locks	Unnecessary	Essential
Invariants	Must hold on entry/exit to data's methods only	Must hold anytime the protecting lock is not held
Deadlock	Impossible	Possible anytime there are multiple unordered locks
Testing	Code coverage finds most bugs, stress testing proves quality	Code coverage insufficient, races cause hard bugs, and stress testing gives only probabilistic comfort
Debugging	Trace execution leading to failure; finding a fix is generally assured	Postulate a race and inspect code; root causes easily remain unidentified (hard to reproduce, hard to go back in time)

A Final Word on "Truths"

Don't underestimate the programming problem.

The hardware community is building parallel hardware, but do you recognize how hard it is to program?

Don't assume the guy upstream can and will solve the hard problems.

This talk will show ideas on future software directions, but these aren't (yet) proven solutions or shipping products.

Hardware semantics and operations should focus on programmability first, speed second.

In particular, non-sequentially consistent memory models are an enormous source of difficulty for programmers.

See for example "Multiprocessors Should Support Simple Memory Consistency Models," Mark D. Hill, IEEE Computer, August 1998. Affirmed at Dagstuhl 2003.

Concurrency

Truths

Consequences

Futures

Today's Status Quo Isn't Enough

The good, the bad, and the ugly

Problem 1: Free threading

willy-nilly concurrency yields higgedly-piggedly failures
as bad as reentrancy (actually, essentially the same problem)
explicit threading is too low-level

Problem 2: Mutable shared memory + locks

locks are the best we have, but aren't composable
locks are hard for expert programmers to get right
("lock-free" isn't the answer; it's hard for geniuses to get right)

All current mainstream languages' concurrency support

based on threads + locks

Granularities Aren't All the Same

Typical levels of concurrency

Coarse: Out-of-box

e.g., web services
emphasis on messaging and async collaboration
shared memory fairly easy to avoid

Coarse: In-box, in-process

e.g., long-running tasks
emphasis on messaging and async collaboration
shared memory often desirable

Fine: Loop parallel and data parallel

e.g., "for all," operations on arrays and other collections
could be shipped off to GPUs, SQL engines, multicore, etc.
shared data is often the point
messages unlikely to be lightweight enough

Toward an "OO for Concurrency"

What we need for a great leap forward

What: Enable apps with lots of latent concurrency at every level

cover both coarse- and fine-grained concurrency,
from web services to in-process tasks to loop/data parallel

map to hardware at run time ("rightsize me")

How: Abstractions (no free threading, no casual data sharing)

active objects asynchronous messages futures
rendezvous + collaboration parallel loops

How, part 2: Tools

testing (proving quality, static analysis, …)
debugging (going back in time, causality, message reorder, …)
profiling (finding convoys, blocking paths, …)

Concurrency

Truths

Consequences

Futures

Concurrency Tools in 2005 and Beyond

Concurrency-related features in recent products:

- OpenMP for loop/data parallel operations (Microsoft, Intel).
- Memory models for concurrency (Java, .NET, C++).
- New/experimental work: Fortress (Sun), Comega (MSR), software transactional memory research (Microsoft, Intel, …).

The rest of this talk is about futures (pun intended):

- The Concur project is aimed at defining:
  - define high-level abstractions for today's (imperative) languages
  - that evenly support the range of concurrency granularities
  - to let developers write correct and efficient concurrent programs with lots of latent parallelism
  - that can be mapped to actual hardware at run time to reenable the free lunch.
- Active objects, messages, futures, parallel loops, parallel STL algorithms, rendezvous, collaboration, … (Examples will use C++.)

[There's lots of other work going on at MS. This happens to be mine.]

Illustrating a Principle: Codifying Idioms

Example: Double-Checked Locking (DCL).

```
volatile Singleton* instance;
Singleton* GetInstance() {
  if( !instance ) {
    // acquire lock
    if( !instance ) {
      instance = new T;
    }
    // release lock
  }
  return instance;
}
```

- Works on some platforms (incl. Java 5 & Visual Studio 2005). Read the manual carefully.
- Error-prone. Omit *volatile*, program compiles & "works."
- Too low-level. Like coding your own vtables…

Replacing with a higher-level abstraction:

```
Singleton* instance;
Singleton* GetInstance() {
  once {
    instance = new T;
  }
  return instance;
}
```

And allowing this variant:

```
Singleton* instance = new T;
Singleton* GetInstance() {
  return instance;
}
```

- Variables should only be initialized once, so we should require the compiler to implicitly Do the Right Thing.

50,000' View: Sources of Concurrency

1. Active objects with async member function calls.

```
active C c;
c.f();                  // these calls are nonblocking; each method
c.g();                  // call automatically enqueues a message for c
…                       // this code can execute in parallel with f & g
```

2. Async calls/work via active lambdas and futures.

Think: "Writing and enqueueing a work item on the fly."

```
x = active { foo(10) };   // call foo asynchronously
y = active { a->b( c ) }; // evaluate asynchronously
p = active { new T };     // allocate and construct asynchronously
return x.wait() * y.wait() * p.wait()->bar();
```

3. Parallel loops and loop bodies.

```
active for each( Employee e in emps ) { … }   // chunk into work items
for( i = v.begin(); i != v.end(); ++i ) active {   // each iter is a work item
  …
}
```

- Gaining/losing concurrency is explicit with active and wait.

Active Objects and Messages

Nutshell summary:

- Each active object conceptually runs on its own thread.
- Method calls from other threads are async messages processed serially ⇒ atomic w.r.t. each other, so no need to lock the object internally or externally. Default mainline is a prioritized FIFO pump.
- Return values and out parameters are futures (future<T>).
- Expressing thread/task lifetimes as object lifetimes lets us exploit existing rich language semantics.

```
active class C {
public:
  void f() { … }
};
```

```
// in calling code, using a C object
active C c;
c.f();              // call is nonblocking
…                   // this code can execute in parallel with c.f()
```

Futures

Return values are future values:

- Return values (and "out" arguments) from async calls cannot be used until an explicit wait for the future to materialize.

```
// in calling code, using a Calc object
future<double> tot = calc.TotalOrders(); // call is nonblocking
… potentially lots of work …             // parallel work
DoSomethingWith( tot.wait() );           // explicitly wait to accept
```

Why require explicit wait? Four compelling reasons:

- No silent loss of concurrency (e.g., early "logFile << tot;").
- Explicit block point for writing into lent objects ("out" args).
- Explicit point for emitting exceptions.
- Need to be able to pass futures onward to other code (e.g., DoSomethingWith(tot) ? DoSomethingWith(tot.wait())).

Example: Multiple Service Requests

A client sends requests to two different services. Cancel them if a reply is not received from both within a timeout.

```
void Process( active Service1& s1, active Service2& s2, int ms ) {
  future<int> result1 = s1.Operation1();
  future<int> result2 = s2.Operation2();

  wait( (result1 && result2) || timeout(ms) );
  if( result1 && result2 ) {
    …                                   // use result1 and result2
  else {
    result1.Cancel();                   // cancel both
    result2.Cancel();
  }
}
```

Using Futures and Active Lambdas

Active blocks (lambdas) for queueing up work items:

```
x = active { foo(10) };          // call foo asynchronously
y = active { a->b( c ) };        // evaluate asynchronously
p = active { new T };            // allocate and construct asynchronously

… more code, runs concurrently with all three active lambdas …

// there is no waiting until we explicitly say "wait" to use results
return x.wait() * y.wait() * p.wait()->bar();
```

Idioms:

- Calling a sync function asynchronously:

```
active { plainObj.Foo(42) };          // type is future<ReturnType>
```

- Calling an async function synchronously:

```
activeObj.Bar(3.14).wait();           // type is ReturnType
```

- Creating a callback to do something when future materializes.

```
future<int> ret = …;
new active { int i = ret.wait(); DoSomethingWith(i); };
```

"Never Call Unknown Code When…"

"Never lock when invoking methods on other objects…"

- Example adapted from *CPiJ2e* (switching to C# syntax):

```
class Particle {
  protected int x;                                   // this object's
  protected int y;                                   // coordinates
  …
  public void Draw( Graphics g ) {
    int localx, localy;
    lock (this) { localx = x; localy = y; }          // safely take copies
    g.DrawRect( localx, localy, 10, 10 );            // call is outside any locks
  }
};
```

- Active objects already have no need for locking (they are inherently serialized), but still want to avoid *blocking*:

```
  public void Draw( Graphics g ) {
    new active{ g.DrawRect( x, y, 10, 10 ); };       // safely takes copies
  }
```

- Both avoid deadlock, but latter is simpler… and more concurrent.

Prevent Blocking Cycles

Avoid calling unknown code while holding a lock… or from an active object's method.

- This isn't as drastic as it sounds, because of active lambdas. Just queue up another work item:

```
active ref class Shape {
  int x, y;                                   // this object's coordinates
  …
public:
  void AddMyselfTo( SyncHashTable^  h ) {
    … do work …
    // If we call "h->Add(this,42);" synchronously, it will probably
    // cause a cycle back to "this" (for GetHashCode). Instead:
    return active{                                  // queue up a work item
      … optionally do more work …
      h->Add(this->wait(),42);
    };
  }
};
```

Comparison: FX Async Pattern

Design Guidelines base async pattern:

- Split single functions into BeginXxx/EndXxx pairs with intermediate structures and explicit callback registrations.
- Sync API:

```
public RetType MethodName(
  Parameters params,
  OutParameters outParams );
```

- Async API:

```
IAsyncResult BeginMethodName(// caller explicitly calls begin/end
  Parameters params,
  AsyncCallback callback,          // caller must supply callback
  Object state );                  // caller usu. must supply cookie

RetType EndMethodName(             // caller explicitly calls begin/end
  IAsyncResult asyncResult,
  OutParameters outParams );
```

Comparison: FX Async Pattern (2)

Issues:

- Intrusive in API: Burying the async work partly inside the APIs changes/bloats the APIs and can duplicate code.
- Complex for callers: Instead of calling one function, the user must call two functions, manage intermediate state, and write an additional function of his own for the callback.
- Hand-coded pattern, no language support/checking: The pattern may vary; consistency depends on manual discipline. This approach has more potential points of failure where the programmer can go wrong.
  - DG example: *"**Do** ensure that the EndMethodName method is always called even if the operation is known to have already completed. This allows exceptions to be received and any resources used in the async operation to be freed."*

Comparison: Async Pattern + Delegates

```
p

  public static v
    int cookieVa
    // … here, ch
    int result = (
      ((AsyncRes
    Console.Writ
  }
}
```

```
int main() {
  future<int> result = active { sampSyncObj.Square( 42 ) };
  // … do useful work …
  Console.WriteLine("Result is {0}", result.wait());
}
```

OpenMP and Beyond

OpenMP offers many benefits:

- Portable, scalable, flexible, standardized, and performance-oriented interface for parallelizing code.
- Hides many details: Thread team created at app startup, per-thread data allocated when #pragma entered, and work divided into coherent chunks.

But it also has many limitations:

- C and Fortran – not C++, C#, Pascal, Python, VB, etc.
- Outside the language, bolted onto code via #pragmas.
- Only good for simple *for* loops over arrays (doesn't work with STL or BCL collections), and for parallel code sections.
- Doesn't take advantage of modern languages' superior abstractions for type safety or generic programming.

Active Loop Bodies

Loop body containing only an active lambda is parallel:

- Loop control statement is evaluated on the master thread.
- Master dynamically assigns loop iterations to team threads.
- Number of threads actually used is tuned to local hardware.

```
for( init; condition; incr ) active {
  // this code is executed across the team
}
for each( Type t in collection ) active {
  // this code is executed across the team
}
while( condition ) active {
  // this code is executed across the team
}
do active {
  // this code is executed across the team
} while( condition );
```

Active Loops

Entire "for each" loop marked active will divide work:

- Loop control statement has to be regular, not customizable.
- Divides work into chunks for master+team threads.
- Number of threads actually used is tuned to local hardware.

```
active for each( Type t in collection ) {
  // each team thread gets a chunk of loop iterations
}
```

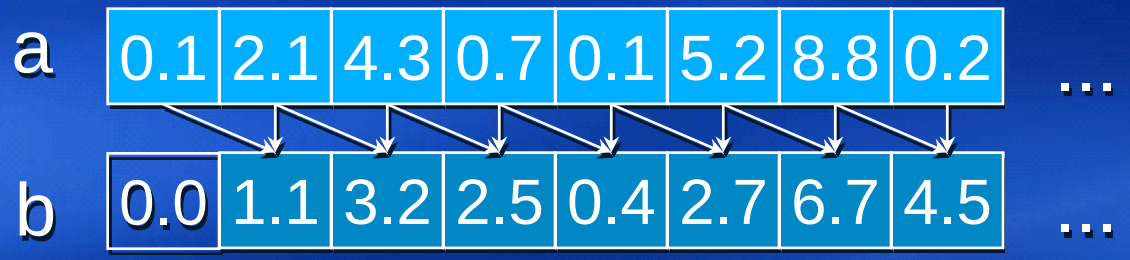
- If there are side effects, use futures to merge results:

```
vector<future<…>> results =
  active for each( Type t in collection ) {
    // each team thread gets a chunk of loop iterations
  };
for each( future<…> r in results ) {
  // do something with r.wait()
}
```

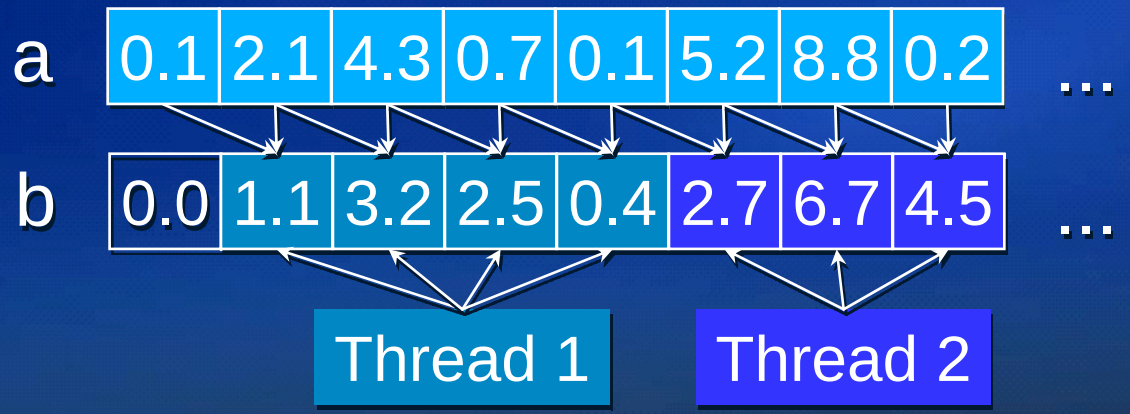
Example: Average-Neighbors

Non-concurrent code:

```
for( i=1; i < n; ++i )
  b[i] = (a[i] + a[i-1]) / 2.0;
```



What we'd like to be able to do:



Example: Average-Neighbors

OpenMP code (example a1):

```
#pragma omp parallel for
for( i=1; i < n; ++i )
  b[i] = (a[i] + a[i-1]) / 2.0;
```

Explicit threads code:

```
DWORD ThreadFn( VOID* pData ) {          // primary function
  for( int i = pData->Start; i < pData->Stop; ++i )
    b[i] = (a[i] + a[i-1]) / 2.0;
  return 0;
}
```

```
for( int i=0; i < n; ++i )               // create thread team
  hTeam[i] = CreateThread( 0, 0, ThreadFn, pDataN, 0, 0 );
WaitForMultipleObjects( n, hTeam, TRUE, INFINITE );
                                         // wait until done
for( int i=0; i < n; ++i )               // clean up
  CloseHandle( hTeam[i] );
```

Example: Average-Neighbors

OpenMP code (example a1):

```
#pragma omp parallel for
for( i=1; i < n; ++i )
  b[i] = (a[i] + a[i-1]) / 2.0;
```

Concur code:

```
active for each( int i in range(1,n) ) {
  b[i] = (a[i] + a[i-1]) / 2.0;
}
```

- Works also with STL and BCL collections, not just arrays.
- Or use a parallel version of the std::transform algorithm.

```
ptransform( a.begin()+1, a.end(), a.begin(), b.begin(), (_1+_2) / 2.0 );
```

first input range

second input range

output range

how to compute the output

Example: Parallel Array Update

OpenMP code:

```
void a7( float *x, int *y, int n ) {
  float a = 0.0;
  int b = 0, i;
  #pragma omp parallel for reduction(+:a) reduction(^:b)
  for( i=0; i<n; i++ ) {
    a += x[i];
    b ^= y[i];
  }
}
```

Concur code:

```
void a7( vector<float> &x, vector<int> &y ) {
  float a = paccumulate( x, 0.0, _1 + _2 );
  int   b = paccumulate( y, 0,   _1 ^ _2 );
}
```

- Also, may get better cache performance by going through one collection and then the other. (The original code may have bad cache behavior unless x and y are related in some way.)

Concurrency

Truths

Consequences

Futures

The First Slide Last

What you need to know about concurrency

It's here

parallelism has long been the "next big thing"

the future is now, and inescapable

It will directly affect the way we write software

the free lunch is over – for sequential apps

only apps with lots of latent concurrency get the free lunch back

languages won't be able to ignore it and stay relevant

The software industry has a lot of work to do,

and we suspect the HW industry vastly underestimates that

a generational advance >OO to move beyond "threads+locks"

key: incrementally adoptable extensions for existing languages

Questions?